ChatGPT

Executive Summary

This **30-day GenAI Architect curriculum** transforms the SocialEagle.ai GenAI Architect outline into an **instructor-ready, cybersecurity-focused training** using primary/official sources. Over 30 days, learners go from AI/ML basics to designing and deploying a secure, enterprise-grade GenAI system for SOC use-cases. The program is hands-on: each day has learning objectives, lecture notes, and a step-by-step lab. We build incremental projects culminating in a final capstone (an "AI SOC Analyst" platform). Key topics include Python & APIs, LLMs & Prompt Engineering, RAG & Vector Databases, Agentic workflows (LangChain, AutoGen, LiveKit), and deployment/security best practices. The course emphasizes **security and governance** (NIST AI RMF, OWASP GenAI Top 10, India's DPDP Act, CERT-In) throughout.

Highlights:

- **Daily structure:** Objectives, short lecture points, hands-on lab tasks with expected outputs, and short quizzes.
- **Tools & code:** Sample Python snippets for ChatGPT/Claude API calls, LangChain chains, vector DB usage, n8n webhooks, and agent code (LangChain/AutoGen).
- **Projects:** Three mini-projects (e.g., SOC Alert Summarizer, RAG Chatbot, Compliance Assistant) and a capstone ("AI SOC Analyst"). Each has clear deliverables.
- **Security/Governance:** Checklists for prompt-injection defense, data privacy, NIST/AWS/Azure controls, and Indian DPDP Act compliance are included, with references (OWASP GenAI Top 10【31†L128-L137】, NIST AI RMF【33†L153-L160】, DPDP Act【45†L58-L67】).
- **Deployment:** We compare cloud options (AWS Bedrock/SageMaker, Azure OpenAI/ML, Google Vertex AI) and self-hosted (HuggingFace, Ollama) in tables, with cost-control strategies (e.g. model choice, batching【50†L848-L856】) and monitoring (logging, LangSmith).
- **Visual aids:** Mermaid diagrams illustrate the 30-day timeline and the GenAI system architecture. Sample UI layouts (Streamlit) are described with code templates.

This curriculum ensures graduates not only **know how to code a GenAI app**, but can **architect it securely, scale it, and align it with business/compliance needs**—exactly the GenAI Architect skillset.

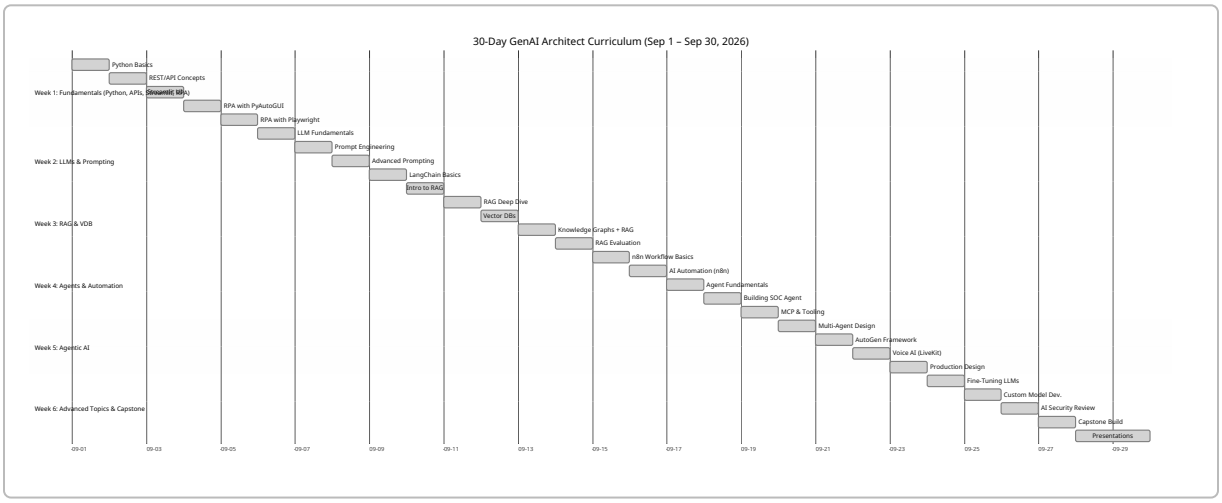
Prerequisites and Setup

- **Prerequisite knowledge:** Intermediate Python (data structures, functions, OOP). Basic understanding of web APIs and cloud concepts. SOC/cybersecurity experience (alerts, SIEM) is a plus but not mandatory.
- **Software/Tools:**
- **Python 3.10+** (Anaconda or venv) with libraries: `openai` / `anthropic` SDKs, `langchain`, `pandas`, `numpy`, `requests`, `playwright`, `streamlit`, `n8n-client` or `requests`.
- **Accounts/API keys:** OpenAI (GPT-4), Anthropic (Claude), Pinecone (or alternative vector DB), and accounts for chosen cloud (AWS/Azure/GCP) and n8n (cloud or local). (Learners may use free tiers/ sandbox where possible.)
- **Dev environment:** VS Code or PyCharm, Git/GitHub for version control, Docker (for advanced labs).

- **Time commitment:** ~2–3 hours/day (including lecture, reading, and lab).
- **Success criteria:** By Day 30, each learner should:
- Build and demo a **working GenAI SOC toolchain** (LLM + RAG + Agents + UI).
- Clearly explain system design trade-offs (model choices, cost vs accuracy, security controls).
- Implement at least one end-to-end project (capstone) and two mini-projects.
- Pass day-end quizzes to show comprehension of key concepts (LLM mechanics, RAG pipeline, security threats, etc.).

Curriculum Overview (30 Days)

Below is the **30-day plan** grouped by themes. Each day lists **Learning Objectives**, brief lecture notes (with references to official sources), a **Hands-on Lab** with step-by-step tasks and expected output, plus a short **Quiz**. We also include **code snippets** for key tasks and **tables** comparing tools.



Week 1: Foundations (Days 1–5)

Day 1: AI, ML & GenAI Fundamentals

Objectives: Understand AI vs ML vs Deep Learning vs GenAI; how LLMs work (transformer, tokens, context window); generative AI use-cases.

Lecture Outline:

- **AI/ML vs Deep Learning:** (No heavy math) ML is about learning patterns from data (supervised/unsupervised), deep learning uses neural nets. Generative AI (GenAI) specifically produces new content (text, code, images).
- **Transformer & LLMs:** Introduce transformer architecture's key concepts: self-attention, layers, tokens (usually 1–4 characters per token). The context window is how many tokens the model can process at once. (Current LLMs: GPT-4 up to 32K+, others vary.)

- **Inference vs Training:** Models are pre-trained on massive text corpora; **inference** is using the model to generate outputs. Fine-tuning means re-training on domain data (rarely needed for retrieval-based use).
- **LLM Use Cases in SOC:** e.g., summarizing logs/alerts, translating IR docs, answering policy questions. Emphasize the **probabilistic** nature: outputs may hallucinate if data is insufficient 【28†L85-L94】, so trust must be earned via context.
- **Hallucinations & Safety:** Define hallucination (model confidently outputs incorrect info)【28†L87-L96】, and mention safety (prompt injection, bias). We will cover security threats later with OWASP/Cert references【31†L128-L137】【47†L268-L275】.

Key Point (RAG intro): Without grounding, LLMs "use probabilities" and can hallucinate or be outdated 【28†L85-L94】【28†L108-L116】. RAG (Retrieval-Augmented Generation) addresses this by supplying relevant external context.

Lab 1: *First LLM API Call*

- **Setup:** Install `openai` Python library and set environment variable `OPENAI_API_KEY`. (Alternatively use Anthropic's `anthropic` lib and `ANTHROPIC_API_KEY`.)
- **Task:** Write a Python script that sends a basic prompt to an LLM:

```
import os, openai
openai.api_key = os.getenv("OPENAI_API_KEY")
resp = openai.ChatCompletion.create(
    model="gpt-4o",
    messages=[
        {"role": "system", "content": "You are a helpful cybersecurity
assistant."},
        {"role": "user", "content": "Explain what a SOC analyst does in
simple terms."}
    ]
)
print(resp['choices'][0]['message']['content'])
```

- **Expected Output:** The model's explanation of a SOC analyst's role. (E.g., "A SOC analyst monitors security systems..."). The output will be a paragraph in plain English.
- **Discussion:** Note how changing `model=` (e.g. to `gpt-3.5-turbo` or `claude-3`) affects style and length. Consider context window: with short prompts it's fine, but large docs will need splitting.

Quiz Q1: *What is a "token" in an LLM context?*
A1: A token is a piece of text (often a word or subword) the model processes; models have a maximum token window. (E.g. "unicorn" → "uni", "corn").

Day 2: Python for AI Developers

Objectives: Solidify Python skills for GenAI tasks: data structures, JSON handling, HTTP requests, virtual env.

Lecture Outline:

- **Python Refresher:** Briefly review lists, dicts, classes. Emphasize working with JSON (gen AI APIs use JSON for prompts/results).
- **Modules & Env:** Using `pip`, virtualenv/conda, `.env` for secrets.
- **Requests & APIs:** How to make HTTP calls (use `requests` lib) to REST APIs (OpenAI, Claude, LangChain endpoints).
- **Data Handling:** Read/write text files, parse JSON responses, handle Pandas if needed.
- **Security Note:** Never hardcode API keys or secrets. Use environment variables and secret managers. (We'll cover more on secrets later.)

Lab 2: *API Integration Practice*

- **Task 1:** Write a Python function that reads a JSON file of a security alert (e.g., `alert.json` containing fields like `ip`, `port`, `failures`) and prints a summary.
- **Task 2:** Use `requests` to call a public API (e.g., `https://api.ipify.org?format=json`) and print the returned IP.
- **Expected:** Correct JSON parsing and printed values.

Quiz Q2: *Which Python data type best holds a parsed JSON object?*
A2: A Python `dict` (dictionary) holds JSON key-value data after using `json.loads()` or `response.json()`.

Day 3: Working with REST APIs

Objectives: Understand REST (GET/POST), JSON, authentication headers; build a small API with Flask or FastAPI; call LLM via an API endpoint.

Lecture Outline:

- **REST Basics:** Explain GET/POST/PUT/DELETE, endpoints, headers, status codes. Use e.g. Flask/FastAPI to illustrate.
- **OpenAI API (RESTful):** Show OpenAI's API docs (JSON payload) to call `POST /v1/chat/completions`. Similarly for Claude (Anthropic) or Azure OpenAI.
- **Authentication:** API keys in headers (e.g., `Authorization: Bearer ...`). Mention rate limits and error handling (4XX, 5XX).
- **Example:** Describe an example API flow: user sends JSON to `/analyze_alert`, backend calls LLM API, returns structured result.

Lab 3: *Build a Simple REST API*

- **Task:** Using Flask or FastAPI, create an endpoint `/analyze_alert` that:
- Receives a JSON payload like `{"alert": "Brute force SSH from 10.0.0.5"}`.
- Calls the LLM (ChatGPT) to analyze it (e.g. "Classify severity and suggest next steps").

- Returns the LLM's JSON response to the client.

- **Sample Code:** (Simplified Flask example)

```
from flask import Flask, request, jsonify
import openai, os
app = Flask(__name__)
openai.api_key = os.getenv("OPENAI_API_KEY")
@app.route("/analyze_alert", methods=["POST"])
def analyze_alert():
    data = request.json
    alert = data.get("alert", "")
    response = openai.ChatCompletion.create(
        model="gpt-4o",
        messages=[
            {"role":"system","content":"You are a SOC assistant."},
            {"role":"user","content": f"Analyze alert:
{alert}. Return JSON with severity and recommended action."}
        ]
    )
    return jsonify(response.choices[0].message.content)
if __name__=="__main__":
    app.run(port=8000)
```

- **Expected:** Sending `POST /analyze_alert` with `{"alert":"..."}` should return a JSON string like `{"severity":"High", "next_steps":"..."}`.

Quiz Q3: *What HTTP method is typically used to send data to create or process resources?*
A3: `POST` is used to send data in the request body (e.g., to create or process a resource).

Day 4: Streamlit Dashboards

Objectives: Learn Streamlit basics to build an interactive UI for GenAI apps (input fields, chat interfaces, display tables).

Lecture Outline:

- **Why Streamlit:** Quickly prototype interactive apps with Python (no frontend code needed)【42†L69-L77】.
- **Widgets:** TextInput, Buttons, FileUploader, st.chat, st.dataframe for tables/charts.
- **State Management:** Session state for chatbots (so conversation persists)【42†L108-L118】.
- **Deployment:** Note ease of deploying on Streamlit Cloud or Snowflake (free for simple apps)【42†L99-L108】.

Lab 4: *Build an AI SOC Dashboard (Streamlit)*

- **Task:** Create a Streamlit app (`app.py`) that:
- Has a **text input** for a security alert message.
- A **Submit** button that sends this input to the Flask API from Day 3 (use `requests.post` internally).
- Displays the JSON response (severity, action) in the app (as text or st.json).

- (Bonus) Maintain chat history by appending to `st.session_state`.

- **Example Code Snippet:**

```
import streamlit as st, requests
st.title("AI SOC Assistant")
alert = st.text_input("Enter security alert")
if st.button("Analyze"):
    if alert:
        res = requests.post("http://localhost:8000/analyze_alert",
json={"alert":alert})
        result = res.json()
        st.json(result)
```

- **Expected UI:** A simple page with an input box. When you type an alert and press Analyze, the returned JSON (e.g. `{"severity":"Medium", "action":"Investigate IP"}`) is displayed nicely.

Quiz Q4: *Name one advantage of using Streamlit for AI apps.*
A4: It allows rapid development of interactive Python apps without HTML/JS (all frontend handled by Streamlit)【42†L100-L109】.

Day 5: Robotic Process Automation (RPA)

Objectives: Learn two RPA tools: PyAutoGUI (desktop automation) and Playwright (browser automation); automate a task and integrate with GenAI.

Lecture Outline:

- **PyAutoGUI:** Controls mouse/keyboard on desktop (screenshots, hotkeys). Good for automating local tasks (e.g. opening an email client to fetch alerts).
- **Playwright:** Browser automation for web tasks (scraping, form-filling). More powerful and scriptable for web apps.
- **Use in GenAI:** E.g., auto-fetch latest logs from a web console, feed into LLM, paste outputs.
- **Examples:** Brief demo of PyAutoGUI clicking icons; Playwright logging into a site and scraping text.

Lab 5: *RPA + AI Integration*

- **Task:** Automate a workflow using Playwright and OpenAI:
- Use Playwright to open a mock web portal (e.g. a local HTML or a public site), log in, and extract some text (e.g. latest alert line).
- Send this text to the ChatGPT API (like Lab 1), parse response.

- Use PyAutoGUI to display a notification or write the result into a local file/email.

- **Step-by-step:** Install `playwright`, run `playwright codegen <site>`, inspect output, then integrate with Python to `await page.textContent("...")`.

- **Expected Output:** The bot prints or notifies "Most severe alert: XX (Attack Type: YY)".

Quiz Q5: *What is a key difference between PyAutoGUI and Playwright?*
A5: PyAutoGUI automates desktop apps (mouse/keyboard), while Playwright automates web browsers (DOM actions, headless mode).

Week 2: LLMs & Prompt Engineering (Days 6–10)

Day 6: Deep Dive into LLMs

Objectives: Understand **how LLMs work under the hood**: tokens, embeddings, attention, temperature/sampling, few-shot vs. fine-tuning.

Lecture Outline:

- **Tokenization & Embeddings:** LLMs convert text to numeric token IDs, then embeddings (dense vectors). Embeddings capture semantic meaning (e.g. "breach" and "hack" close in vector space).
- **Attention Mechanism:** Explain self-attention briefly: each token "attends" to others to build context. Larger models just stack more layers to capture complex patterns.
- **Decoding Parameters:** *Temperature* (controls randomness: higher = more creative) and *Top-p/top-k* (sampling controls). Low temperature + top-p=1 yields deterministic answers.
- **Prompting vs Fine-tuning:** Retraining a model (fine-tuning) vs simply writing good prompts. OpenAI suggests **embeddings + RAG instead of fine-tuning for knowledge**【5†L15-L21】. (It's cheaper and more flexible.)
- **Leading Models (2026):** Quick survey: GPT-4/GPT-4o (OpenAI), Gemini (Google), Claude (Anthropic), open-weight (Llama3/Mistral). Discuss trade-offs (context size, multimodal support, enterprise features)【19†L117-L126】【19†L144-L151】. Key point: *"No single model is best for every use case"*【19†L51-L59】.

Key Point: LLM selection must factor context length, latency, multimodal needs, and compliance【19†L117-L126】【19†L144-L151】. For example, large context windows aid lengthy incident analysis, but come with higher compute.

Lab 6: *Experiment with LLM Behavior*

- **Task 1:** Compare outputs of two models: Use GPT-4 (OpenAI) vs Claude-3 (Anthropic) on the same prompt, e.g. "Explain port scanning to a new SOC analyst." Observe differences in style/length.
- **Task 2:** Practice **few-shot prompting**: Craft a prompt with 2–3 examples of alerts with their classification, then ask the model to classify a new alert. Observe improved accuracy.
- **Expected:** GPT-4 might give bullet-list answer, Claude may give more narrative. Few-shot improves output consistency (it follows example format).

Quiz Q6: *Why might you choose GPT-4o (OpenAI) over Llama3 (open-source) in an enterprise?*
A6: GPT-4o likely has higher accuracy and enterprise support (managed service), whereas Llama3 (open) needs self-hosting and may have smaller context. (See LLM criteria【19†L117-L126】【19†L144-L151】.)

Day 7: Prompt Engineering Principles

Objectives: Learn the **RTCFR** framework for prompts: **Role, Task, Context, Few-shot, Response format**. Understand system vs user prompts, delimiting context.

Lecture Outline:

- **Prompt Components:**
- *Role/System Instruction*: e.g. "You are a SOC analyst specializing in intrusion detection."
- *Task/User Instruction*: e.g. "Analyze the following alert."
- *Context/Data*: the actual alert data or additional info.
- *Few-shot Examples (if any)*: showing format.
- *Expected Output Structure*: JSON, bullet list, etc.
- **Examples:** Show simple vs structured prompts. E.g., unstructured "What's this alert mean?" vs structured with JSON output.
- **JSON/Structured prompting:** Always specify exact JSON schema if you need machine-parsable output【9†L1425-L1433】. For instance:

```
{"severity": "HIGH", "attack_type": "Credential Stuffing", "confidence": 0.92}
```

- **Avoiding Ambiguity:** Use delimiters (`` ``` `` or quotes) around user data to prevent it being interpreted as instructions.
- **Pitfalls:** Be aware of **prompt injection**: attackers may attempt to insert instructions into user data. (OWASP LLM01 says "Prompt Injection" is a top risk【31†L128-L137】.) Techniques like context separation, content filters, and output validation are needed (more on Day 29).

Lab 7: *Craft and Test Prompts*

- **Task:** For a given security alert text, design prompts of different styles:
- **Zero-shot:** "Analyze alert: <alert_text>"
- **Few-shot:** Provide 2 example alerts with desired JSON answers, then a third alert to analyze.
- **Role-based:** Add "You are a security expert." at the start.
  Use the LLM to run all prompts and compare accuracy.
- **Expected:** The role-based and few-shot prompts typically yield more consistent structured answers. Zero-shot may be vague.

Quiz Q7: *What is "prompt injection" and why is it dangerous?*
A7: Prompt injection is when malicious input (e.g. in `alert_text`) includes instructions that change the model's behavior. It can bypass filters or cause the model to reveal or alter information【31†L128-L137】.

Day 8: JSON & Template Prompting

Objectives: Learn how to get **LLMs to output JSON** reliably, and use prompt templates in LangChain.

Lecture Outline:

- **JSON Output:** Modern LLMs (GPT-4, Claude) can output JSON if properly instructed. Always start with a clear instruction like *"Answer in valid JSON only."* and provide a schema example【9†L1551-L1560】.
- **Prompt Templates (LangChain):** These allow you to define templates with variables. E.g. using `PromptTemplate`:

```
from langchain.prompts import PromptTemplate
template = "Analyze alert: {alert_text}\nReturn JSON with keys severity,
type, confidence."
prompt = PromptTemplate(input_variables=["alert_text"], template=template)
```

This ensures consistent formatting.

- **Validation:** Always validate LLM JSON outputs (e.g. with Python `json.loads()`) and handle errors if it's malformed. Use tools like regex or output parsers in production.

Lab 8: *JSON Prompting with LangChain*

- **Task:** Use LangChain's `LLMChain` or `PromptTemplate` to prompt the model and parse output.

```
from langchain import LLMChain, OpenAI
from langchain.prompts import PromptTemplate
template = """You are a SOC assistant.
Analyze the following alert and output JSON with keys: severity,
attack_type, confidence.
Alert: {alert}"""
prompt = PromptTemplate.from_template(template)
chain = LLMChain(llm=OpenAI(model_name="gpt-4o"), prompt=prompt)
result = chain.run({"alert": "Multiple failed SSH logins from 10.1.2.3"})
print(result)
```

- **Expected:** The output should be a JSON-like string, e.g.:

```
{"severity": "Medium", "attack_type": "Brute Force", "confidence": 0.88}
```

You may need to manually `import json` and fix format if it prints quotes incorrectly.

Quiz Q8: *In prompt templates, what is the purpose of an* `input_variables` *list?*
A8: It tells the template which placeholders (e.g. `{alert}`) to replace with user-provided values. LangChain uses these to fill the prompt.

Day 9: Introduction to LangChain (Chains & Retrieval)

Objectives: Learn LangChain's core abstractions: LLMs, Prompts, Chains, Retrievers, Tools.

Lecture Outline:

- **LangChain Overview:** LangChain is a Python library for building LLM applications【7†L131-L139】. Key concepts:
- **LLM wrapper:** e.g. `OpenAI(model="gpt-4o")` or `Anthropic(...)` to interact with models.
- **PromptTemplate:** Parameterized prompts (as above).
- **Chain:** A sequence of steps (LLM + parsing, or multi-turn conversation). For instance, `LLMChain` ties an LLM to a prompt template.
- **ConversationalChain:** Manages memory and chat history.
- **Retrievers:** Abstraction for search. Typically built on vector DBs. (We'll use them in RAG). Example: `Retriever.from_vectorstore(vector_store)` in LangChain gives an object that can `get_relevant_documents(query)`.
- **Tools & Agents:** (Teaser) Tools are functions (e.g. a weather tool, or our `search_documentation` in [9†L1521-L1530]). Agents orchestrate calling LLM and tools.
- **Example Chain:** Show pseudocode:

```
from langchain.chat_models import ChatOpenAI
from langchain.chains import ConversationChain
llm = ChatOpenAI(model="gpt-4o")
chain = ConversationChain(llm=llm, verbose=True)
print(chain.run("Hello, who is my SOC manager?"))
```

This maintains chat state.

- **Why LangChain:** It helps structure complex interactions (multi-step reasoning, RAG) without writing plumbing code.

Lab 9: *Simple LangChain Chat*

- **Task:** Using LangChain, create a simple chat loop.

```
from langchain.chat_models import ChatOpenAI
from langchain.chains import ConversationChain
llm = ChatOpenAI(model="gpt-4o")
chat = ConversationChain(llm=llm, verbose=False)
user_input = input("User: ")
while user_input.lower() != "exit":
    response = chat.run(user_input)
    print("Bot:", response)
    user_input = input("User: ")
```

- **Expected:** You can type messages like "Hi, classify this alert..." and see the LLM reply. The chain maintains context (e.g., follow-ups).

Quiz Q9: *What is a "Retriever" in LangChain?*
A9: A Retriever is a component that, given a query, returns relevant documents or text chunks (often via semantic search). It bridges to RAG by fetching context for the LLM.

Day 10: RAG Fundamentals

Objectives: Grasp **Retrieval-Augmented Generation (RAG)** concept, components (vector DB, embeddings, chunking), and benefits.

Lecture Outline:

- **What is RAG?** RAG uses external data to "ground" model responses【28†L108-L117】. The four steps: *Ingestion*, *Retrieval*, *Augmentation*, *Generation*【28†L108-L117】.
- **Why RAG:** Improves accuracy by giving LLM the **right context**. Allows private/proprietary data use (models don't "know" your data by default)【28†L110-L119】. Builds trust via citations and real data. Also often *cheaper than fine-tuning*【28†L133-L142】.
- **Components:**
- *Document Loader/Splitter:* Break large docs/logs into chunks (e.g. 1000 tokens) using `RecursiveCharacterTextSplitter`【9†L1503-L1512】.
- *Embeddings:* Convert chunks to vectors (OpenAI, Cohere, etc).
- *Vector DB:* Store embeddings for similarity search (Pinecone, Weaviate, etc).
- *Retriever:* Query index with user question to get top-K relevant chunks.
- *Context Injection:* Combine retrieved texts into the prompt for the LLM. Could be appended or fed to an agent (like LangChain's `RetrievalQA` chain).
- **RAG vs Fine-tuning:** If knowledge changes often or is large, RAG is preferred【5†L5-L8】. Fine-tuning is more static and expensive.
- **RAG + Agents:** Mention that agents can orchestrate RAG steps and **validate sources**【28†L149-L158】. (Agents can score or reject retrieved info, ask follow-up queries, etc.)

Lab 10: *Basic RAG Pipeline with LangChain*

- **Data:** Download or use sample security docs (e.g., MITRE ATT&CK wiki or a PDF of company policies).
- **Tasks:**
- **Chunk & Embed:** Use LangChain's `RecursiveCharacterTextSplitter` to split documents, and `OpenAIEmbeddings()` to embed them.
- **Vector Store:** Initialize a vector store (e.g. Pinecone or Chroma) and add embeddings. Example (Chroma):

```
from langchain.vectorstores import Chroma
embeddings = OpenAIEmbeddings()
vector_store = Chroma.from_texts(docs, embedding=embeddings,
collection_name="sec_docs")
```

(Cite: the LangChain docs example [9†L1548-L1557]).

- **Query:** Write a function that takes a natural question (e.g. "What controls mitigate phishing?"), performs `vector_store.similarity_search(query, k=3)`, and prints the top 3 chunks.
- **LLM Answering:** Feed these chunks plus the question into the LLM prompt to answer. (Use prompt: "Based on the following docs: [chunks], answer: ...").
- **Expected:** E.g., for "What controls mitigate phishing?", retrieved snippets about awareness training, email filtering should appear. The LLM answer should cite or reflect those controls.

Quiz Q10: *Name one benefit of RAG over just using a model's training data.*
A10: RAG can use up-to-date or proprietary data, allowing context-aware answers (and even citations). It grounds responses in real data【28†L133-L142】.

Week 3: Advanced RAG & Knowledge Graphs (Days 11–15)

Day 11: RAG Architectures

Objectives: Explore **RAG patterns** and components in depth; multi-step retrieval & analysis.

Lecture Outline:

- **RAG Variants:** Simple vs. **Deep RAG**. Simple RAG: retrieve docs, concatenate, ask LLM. Deep RAG: use agents/subagents for analysis (as in LangChain DeepAgents tutorial【9†L1539-L1548】【9†L1552-L1561】).
- **Example Pattern:** (From LangChain DeepAgents) Index docs, search, then delegate chunk analysis to sub-agents, and synthesize【9†L1539-L1567】. This keeps the "orchestrator" prompt small and offloads work.
- **RAG Considerations:**
- *Chunk Size:* Too big = context overflow; too small = lose context. Aim ~500–1000 tokens.
- *Retrieval K:* More chunks (larger K) = more recall, but more prompt cost.
- *Citation:* In output, cite sources (e.g. "(Doc1)") to increase trust.
- **Evaluation:** Ground truth queries: check if answers use retrieved info (faithfulness). Track precision/recall of retrieval separately.

Quiz Q11: *What is a trade-off when setting a large K (number of retrieved chunks) in RAG?*
A11: Larger K may increase answer completeness (recall) but costs more tokens (higher API cost) and may hit context limits.

Day 12: Vector Databases Comparison

Objectives: Compare major vector DBs (Pinecone, Weaviate, Qdrant, Chroma, FAISS, pgvector) for RAG.

Lecture Outline:

- **Managed vs Self-hosted:** Pinecone, Weaviate Cloud, Qdrant Cloud = managed (easy, auto-scale)【21†L298-L307】. Milvus, Weaviate OSS, Qdrant OSS = open-source (more control, free but require ops). Chroma = lightweight embedded (great for prototyping【22†L37-L45】).

- **Key Features:**
- *Hybrid Search:* Weaviate & Qdrant support combined vector+keyword filters natively【21†L249-L258】.
- *Scalability:* Milvus/Pinecone can handle billions of vectors (Milvus is OSS, Pinecone is paid). Chroma/pgvector better for <10M【22†L31-L39】.
- *Namespaces/Tenancy:* Pinecone offers multitenant namespaces (up to 100k)【21†L252-L261】. Useful for SaaS.
- *Index types:* HNSW, IVF, PQ (most use HNSW for approximate NN) – we won't dive deep here, but be aware.
- **Cost Considerations:**
- Pinecone charges by storage+ops【21†L298-L307】. Weaviate OSS is free but you host it.
- Chroma's free (Apache 2.0) but limited to prototypes【22†L39-L47】.
- You can often start with Chroma or Redis-backed vector stores, then move to production DB as needed【21†L323-L331】.
- **Tool Comparison Table:**

Database	Type	Best Use Case	Key Strengths	Key Limitations
Pinecone	Managed SaaS	10M–100M+ vectors	Serverless, auto-scaling, multi-tenant	Usage-based pricing【21†L298-L307】
Weaviate	OSS + Cloud	RAG with filters	Hybrid semantic+keyword search, context graph support【26†L121-L129】	Cloud paid tier beyond trial, self-host complex
Qdrant	OSS + Cloud	RAG & multi-modal	Good free tier, GPU acceleration, geo support	Open-source ops overhead
ChromaDB	OSS (embedded)	Prototyping (<10M)	Developer-friendly API, embedded use【22†L45-L54】	Not for large-scale (per docs)【22†L55-L64】
Redis/ pgvector	Plugin/ Extension	Existing SQL/ Redis stack	Use existing DB, easy for small scale	Limited to CPU, scaling complex
FAISS	OSS (library)	On-prem local search	Highly optimized (e.g. HNSW) for small clusters	No built-in multi-user or persistence (need work)

(Sources: Firecrawl vector DB analysis【21†L298-L307】【21†L333-L341】【22†L31-L39】; Weaviate official docs【26†L121-L129】.)

Lab 12: *Hands-on with a Vector DB (Chroma)*

- **Task:** Use `langchain.vectorstores.Chroma` for a quick test (since no API key needed).

```
from langchain.vectorstores import Chroma
from langchain.embeddings.openai import OpenAIEmbeddings
```

```
docs = ["Phishing is sending fraudulent emails", "Malware is malicious
software", "Use strong passwords and MFA"]
embeddings = OpenAIEmbeddings()
vectordb = Chroma.from_texts(docs, embedding=embeddings,
collection_name="security")
results = vectordb.similarity_search("How do I protect accounts?", k=2)
print(results)
```

- **Expected:** It should retrieve chunks relevant to "protect accounts", likely "Use strong passwords and MFA".

Day 13: Advanced RAG Techniques

Objectives: Learn production RAG tricks: query expansion, metadata filtering, reranking.

Lecture Outline:

- **Query Rewriting:** Before search, rephrase user query using LLM (as a "query enhancer") to improve retrieval.
- **Metadata Filters:** If documents have metadata (e.g. source, date), filter on those (e.g. only "Internal Reports"). Weaviate makes this easy via GraphQL or API.
- **Hybrid Search:** Combining vector similarity with keyword matching (Weaviate does this natively 【21†L249-L258】). E.g. ensure certain terms are present.
- **Reranking:** After retrieving top-K, use a separate model (or the LLM itself) to re-score or prune them. LangChain has an `EmbeddingChain` to refine context.
- **Caching & Efficiency:** Cache embeddings and search results for frequent queries. Use smaller models for retrieval queries if possible.
- **Hallucination Test:** Always **validate** answers: e.g., do a second pass asking the LLM "Which source did you use? Quote it." to check truthfulness.

Quiz Q13: *What is one benefit of query rewriting in RAG pipelines?*
A13: It refines the user's question (e.g. expanding abbreviations or adding synonyms) so the vector search returns more relevant documents.

Day 14: Knowledge Graphs + RAG

Objectives: Introduce Knowledge Graph (KG) concepts, and how they integrate with RAG for richer context.

Lecture Outline:

- **What is a Knowledge Graph:** Entities (nodes) and relationships (edges). In cybersecurity, e.g. "ThreatActor → uses → Tactic" nodes.
- **KG vs Plain Text:** KGs allow structured queries (graph traversals) and multi-hop reasoning. They complement RAG: use RAG for text retrieval, KG for entity relations.
- **Example (MITRE ATT&CK):** Show a small KG of e.g.:

```
APT29 → uses → Credential Dumping (T1003)
APT29 → targets → Active Directory
```

Such a graph can answer "Which threat actors perform credential dumping?"

- **Hybrid KG-RAG:** Some systems (like Weaviate) combine vector search with graph (metadata/triplets). Alternatively, after retrieving docs, extract entities (SpaCy or LLM extraction) to populate a KG.
- **Tools:** Mention Neo4j or NetworkX for building KGs, or Knowledge Graph APIs. LangChain has a `KnowledgeGraph` chain (not core, but possible).

Lab 14: *Build a Mini Threat Graph*

- **Task:** Using Python, create a simple graph (e.g., `networkx` or just dictionaries). Add nodes for 2–3 adversaries and their techniques from MITRE.

```python
import networkx as nx
G = nx.DiGraph()
G.add_edge("APT28", "Credential Dumping", relation="uses")
G.add_edge("APT28", "Discovery", relation="uses")
G.add_edge("APT28", "Tactic: Initial Access", relation="targets")
G.add_edge("APT29", "Credential Dumping", relation="uses")
# etc.
```

- **Tasks:** Query the graph (e.g., find all techniques APT28 uses). Use `G.predecessors` / `G.successors`.
- **Expected:** For "APT28", list "Credential Dumping, Discovery". For "Credential Dumping", list "APT28, APT29". This illustrates KG power.

Quiz Q14: *Name one advantage of using a Knowledge Graph in threat intelligence.*
A14: KGs allow you to explicitly model relationships (e.g. which attacker uses which technique) and query across them (multi-hop), which plain text search may miss.

Day 15: Evaluating RAG

Objectives: Learn how to **measure RAG performance**: retrieval quality and answer faithfulness.

Lecture Outline:

- **Retrieval Metrics:** Precision@K (of retrieved docs vs ground truth), Recall (how many needed docs were retrieved). You can build a small test set (query + relevant doc IDs).
- **Answer Evaluation:** Check that the LLM's answer is *grounded* in retrieved text. Techniques:
- **Overlap:** Are key facts in answer present in any retrieved chunk?
- **Hallucination Check:** Run the answer through another LLM with prompt "Cite sources or indicate uncertainty."
- **Automated Testing:** Use benchmarks (e.g. LlamaIndex Evaluate, LangSmith Datasets) to track over time.

- **User Testing:** Show answers to SME and gather feedback.

Deliverable for Week 3: AI RAG Chatbot for Security Q&A. (Use previous code to create a Streamlit interface that retrieves from a security doc DB and answers queries.)

Quiz Q15: *What is a "grounded" LLM answer in RAG?*
A15: An answer is grounded if its assertions match the retrieved context and cited sources, not just the model's memory.

Week 4: Automation & AI Agents (Days 16–21)

Day 16: Workflow Automation with n8n

Objectives: Learn n8n basics (nodes, triggers, data flows) and how to call external services.

Lecture Outline:

- **What is n8n:** An open-source workflow automation tool (like Zapier) with a visual editor【11†L21-L29】. Supports "Webhooks" as triggers, and nodes for HTTP requests, data transform, and AI.
- **Key Concepts:** Workflow = series of *nodes*. A webhook trigger node starts a workflow on HTTP POST【11†L31-L42】. "Set" and "Function" nodes can transform data.
- **Integrations:** n8n has built-in OpenAI/Claude nodes (so you don't have to code API calls). It also integrates with Slack, Email, Databases, etc.
- **Use-case:** We'll use n8n to glue together the SOC pipeline: incoming alerts → LLM analysis → ticket creation.

Lab 16: *Create an n8n Workflow*

- **Prerequisite:** Sign up for n8n cloud (or run locally with Docker).
- **Task:** Build a workflow:
- **Trigger:** HTTP Webhook node (copy its URL).
- **LLM Node:** Use the built-in OpenAI node. Connect from Webhook. Map the alert text to the prompt.
- **Output:** Send results to e.g. Slack or Email node. For example, use Slack node to post the LLM's assessment.
- **Step-by-step:** In n8n Editor, add a Webhook node, set as trigger "POST /webhook/analyze". Add "OpenAI" node, connect, set model and prompt (use JSON structure). Add Slack node, connect, and map message.
- **Test:** Send a `POST` to the webhook (via `curl` or Postman):

```
curl -X POST 'https://xyz.webhook.n8n.cloud/webhook/analyze' -H 'Content-Type: application/json' -d '{"alert":"Suspicious login from IP 5.5.5.5"}'
```

- **Expected:** The LLM node returns a JSON with analysis, which is posted to the configured Slack channel. (Alternatively, log to console.)

Quiz Q16: *What advantage does n8n offer for AI workflows?*
A16: n8n lets you visually orchestrate APIs and tools (including LLMs) without writing glue code. It scales to many integrations (Slack, DBs, Git, etc.) easily【11†L21-L29】.

Day 17: AI-Driven Automation

Objectives: Build a security-specific automated workflow (e.g., alert triage pipeline) using n8n and LLMs.

Lecture Outline:

- **Scenario:** New SIEM alert arrives, automate triage.
- **Workflow Steps:** (Example)
- *Webhook Trigger* (external SIEM posts alert JSON).
- *Extract Details* (n8n "Set" node splits JSON fields).
- *LLM Analysis* (calls ChatGPT to classify alert, map to MITRE, score risk).
- *Action:* Based on LLM output, either auto-close low-priority, or create Jira/Ticket (n8n Jira node).
- **Data Flow:** Show how n8n passes JSON between nodes. Each node can run JS code if needed (Function node).
- **Guardrails:** Introduce quick idea of checking LLM's "threat level" before auto-closing. (Manual steps if uncertain.)

Lab 17: *Automated Alert Triage*

- **Task:** Extend yesterday's workflow:
- After LLM output, add a *Function* node to parse severity (e.g., code in n8n: `if (items[0].json.severity == 'Low') return; else return items;` to only continue on Medium/High).
- If risk is Medium/High, use n8n's Jira or GitHub Issues node to create an issue with LLM details.
- (Optional) Send email via SMTP or Slack for notification.
- **Expected:** Posting a test alert of severity "Low" should be ignored. Posting a "High" alert should auto-create a ticket with the summary.

Quiz Q17: *What node type in n8n can you use to execute custom JavaScript?*
A17: The **Function** node allows running arbitrary JS on the incoming item data.

Day 18: Introduction to AI Agents

Objectives: Learn the distinction between simple LLM calls and **agents** (multi-step, tool-using LLM workflows); grasp agent architecture (planner, tools, memory).

Lecture Outline:

- **What is an Agent?** An LLM agent is an orchestrator that **receives a high-level goal and autonomously decides which tools or steps to use**【28†L149-L158】. (E.g., "Investigate IP" could invoke WHOIS, VirusTotal, then consult a knowledge base.)
- **Agent Loop:** Agent cycle: **Plan** (decide next action), **Act** (use a tool or ask LLM), **Observe** (get result), repeat until done.
- **Tools:** Agents require tool definitions. A tool can be an API (e.g. `query_whois(ip)`), or a function (e.g. searching a database).
- **Example:** Tool definitions for SOC:
- `whois_tool(ip)`: calls WHOIS API, returns domain info.
- `vt_tool(ip)`: queries VirusTotal for threats.
- `mitre_tool(term)`: looks up MITRE ATT&CK info.
- **Chain vs Agent:** LangChain's `AgentChain` or Microsoft's AutoGen allow defining agents that pick tools. We will use AutoGen (Microsoft) soon.

Lab 18: *Build a Simple SOC Agent with AutoGen*

- **Setup:** Install Microsoft AutoGen framework (`pip install autogen-agentchat autogen-ext[openai]`).
- **Task:** Write a Python agent that:
- Uses `AssistantAgent` (AutoGen) with OpenAI.
- Has a tool (function) for WHOIS lookup. Example stub:

```
@tool
def whois_lookup(ip: str) -> str:
    # call an actual WHOIS API or use python-whois
    return f"FakeWHOIS data for {ip}"
```

- Runs with a task: e.g., "Investigate IP 8.8.8.8 for security threats."

- The agent should decide to call `whois_lookup`, get result, and produce an answer.
Use the AutoGen example as a template【37†L29-L38】.

- **Sample Code:**

```
import asyncio
from autogen_agentchat.agents import AssistantAgent
from autogen_ext.models.openai import OpenAIChatCompletionClient
from autogen_agentchat.tools import tool

@tool
def whois_lookup(ip: str) -> str:
    return f"WHOIS info for {ip}: (stub)"

async def main():
```

```
    agent = AssistantAgent("sec_agent",
OpenAIChatCompletionClient(model="gpt-4o"))
    agent.add_tools([whois_lookup])
    result = await agent.run(task="Investigate IP 8.8.8.8")
    print(result)

asyncio.run(main())
```

- **Expected:** The agent should mention using `whois_lookup` and include its stub data in reasoning, e.g.:

```
# Plan
1. Use whois_lookup on 8.8.8.8
# Observation
"WHOIS info for 8.8.8.8: (stub)"
# Conclusion
"The IP 8.8.8.8 is owned by Google and is not known for malware, risk is
low."
```

Quiz Q18: *In an agent context, what is a "tool"?*
A18: A tool is a function or API the agent can call (e.g., a WHOIS lookup or database query). Agents use tools to gather facts or take actions during their planning【28†L149-L158】.

Day 19: Building a Security Agent

Objectives: Create a more complex AI agent that chains multiple tools for an investigative task.

Lecture Outline:

- **Toolchain Example:** For IP investigation, tools include: WHOIS, VirusTotal API, internal SIEM query (if accessible), MITRE mapping.
- **Agent Workflow:** The agent might plan: "First run whois, then virus scan, then summarize findings." Show how AutoGen or LangChain agents can loop with tools.
- **Memory in Agents:** Agents can have persistent memory (e.g., store facts found) or output results via MCP (next day).
- **Safety:** Emphasize that agents can make network calls – ensure secure endpoints and not trusting agent with elevated privileges blindly (OWASP LLM10 "excessive agency" risk【31†L163-L172】).

Lab 19: *Multi-Tool Agent for Incident Response*

- **Task:** Extend the Day 18 agent:
- Add a `virus_total(ip)` tool (use `requests` to VirusTotal API or stub).
- Add a `mitre_lookup(technique)` tool (can be a simple `return "Mitre info for " + technique`).

- Give the agent a compound task: “Investigate IP 1.2.3.4, check whois and VT, then identify possible MITRE technique.”

- **Expected:** The agent should use WHOIS and VT, then perhaps call `mitre_lookup` on any found malware family.
  The final print might look like:

```
# Plan:
1. whois_lookup(1.2.3.4)
2. virus_total(1.2.3.4)
3. mitre_lookup("Malware-X")
# Observations:
WHOIS ...
VT: "Malware-X"
MITRE: "Malware-X is Txxxx"
# Answer:
IP has Malware-X, which corresponds to MITRE technique Txxxx (Injection).
```

Quiz Q19: *Why limit an agent’s tools and capabilities in a production system?*
A19: To reduce risks: fewer tools mean lower chance of harmful actions. Principle of least privilege (OWASP LLM08: “Excessive Agency” warns against giving too much autonomy)【31†L163-L172】.

Day 20: Model-Context Protocol (MCP)

Objectives: Understand the Model-Context Protocol (MCP) for enabling agents to interact with external data sources (databases, IDE, etc.) in a structured way.

Lecture Outline:

- **What is MCP:** A protocol where an LLM agent can request actions from external “resources” (like SQL database, file system) as part of its reasoning. The agent sends JSON with `{{action:..., params:...}}`, gets JSON in reply【37†L24-L33】.
- **Architecture:** Usually involves an MCP server that connects the LLM to back-end (e.g., `McpWorkbench` extension in AutoGen【37†L62-L70】). It can allow **tool use over APIs** more flexibly than hard-coded Python tools.
- **Use-cases:** Agents using MCP might say “Query SIEM logs for event X” as an action. The MCP server (like a Python service) executes the SQL and returns results. This separates agent logic (planning) from execution.
- **Security:** Because MCP allows code execution, it must be tightly controlled (auth, sanitization) – see OWASP prompt injection concerns and LLM07 (plugin security)【31†L158-L162】.

Lab 20: *Experiment with MCP (Conceptual)*

- **Note:** Setting up a full MCP is complex; we’ll do a simplified example.
- **Task:** Assume an MCP server where agent can call `search_siem(query)`. In our code, stub the MCP:

```
def search_siem(query):
    return "Found 5 incidents matching query."
```

Then in AutoGen agent, define a tool `@tool` using `search_siem`.

- **Expected:** The agent could `{"action":"search_siem","params":{"query":"malware alerts"}}` and receive the stubbed string.
- **Discussion:** In practice, you'd use AutoGen's `McpWorkbench` to hook a real service.

Quiz Q20: *Name one risk of giving an AI agent external data write access via MCP.*
A20: The agent could inadvertently overwrite or delete data. (So control actions strictly. OWASP LLM07 warns about insecure plugin design.)

Day 21: Agent + MCP Workflow

Objectives: Combine all pieces: architect an agentic workflow that uses MCP or tools to interact with actual systems.

Lecture Outline:

- **System Diagram:** Show architecture (mermaid below) where user input goes to an Agent Orchestrator (LangChain/AutoGen), which calls LLM and external tools via MCP. The orchestrator may loop, then output to a UI or ticketing system.
- **Stream of Tools:** Example pipeline: *User query → Agent → {SIEM, WHOIS, MITRE, Vector DB (RAG)} → Synthesized report.*
- **Multi-Agent vs Single Agent:** Discuss when to split into specialized agents (threat intel agent, risk scoring agent, report agent) vs one monolithic agent.

```
flowchart LR
    User[User Input (e.g. “Investigate IP”)] --> Orchestrator[Agent
Orchestrator]
    Orchestrator --> LLM[LLM Model (GPT-4o / Claude)]
    Orchestrator --> SIEM[SIEM Query Tool]
    Orchestrator --> WHOIS[WHOIS Tool]
    Orchestrator --> MITRE[MITRE Lookup Tool]
    LLM -- uses --> SIEM
    LLM -- uses --> WHOIS
    LLM -- uses --> MITRE
    SIEM -- data --> LLM
    WHOIS -- data --> LLM
    MITRE -- data --> LLM
    Orchestrator --> Dashboard[Streamlit UI / Jira]
```

Architecture of AI SOC Agent: The user's request is sent to an Agent Orchestrator, which queries the LLM. The agent can call various tools (e.g. SIEM, WHOIS, MITRE) via the MCP (or direct tool functions). The orchestrator compiles responses and delivers a final output (on a dashboard, ticket, or report).

Quiz Q21: *In our SOC Agent example, what role does the "Agent Orchestrator" play?*
A21: It acts as the central controller, taking user queries, invoking LLM and tools (via MCP or function calls), and synthesizing the final answer or action.

Week 5: Agentic AI (Days 22–25)

Day 22: Multi-Agent Systems

Objectives: Learn about designing systems of multiple specialized agents rather than one large agent.

Lecture Outline:

- **Why Multiple Agents:** Improves scalability and specialization. E.g., one agent handles "SOC analysis", another "Threat Intel lookup", another "Report generation"【37†L22-L30】.
- **Coordinator/Manager:** Often a top-level agent assigns tasks to sub-agents (like a "manager agent")【37†L49-L58】.
- **Parallelism:** Agents can work concurrently on sub-tasks (like the LangChain DeepAgents example where subagents analyze doc chunks).
- **Example Structure:** (Mermaid style):

```
Manager Agent: "Investigate incident"
  ├─ SOC Agent: query SIEM, gather alerts
  ├─ Intel Agent: query VirusTotal, WHOIS
  ├─ Mitre Agent: correlate with techniques
  └─ Report Agent: compile findings into a summary
```

Show how results flow back to Manager.

- **Coordination:** Agents may communicate (or everything flows up through manager). Multi-agent frameworks (AutoGen Core or LangGraph) support this pattern.

Quiz Q22: *What's one benefit of using multiple smaller agents instead of one big agent?*
A22: Specialization: each agent can focus on a narrow task (improving accuracy), and they can operate in parallel for efficiency【37†L52-L60】.

Day 23: AutoGen Framework (Microsoft)

Objectives: Get hands-on with Microsoft's AutoGen for building multi-agent apps; learn about `AssistantAgent`, `Tool`, and conversation.

Lecture Outline:

- **AutoGen Recap:** It provides `AgentChat` for conversational agents and `Core` for multi-agent workflows【37†L22-L30】【37†L49-L58】.
- **AgentChat (we used):** Good for building a single agent with tools (like Day 18).
- **Core (multi-agent):** Allows defining roles and communication paths. E.g., ManagerAgent can spawn `AssistantAgent` threads.
- **Example Use:** The `agent.run(task=...)` call can be used within Core agents to communicate.
- **Studio & Extensions:** Mention AutoGen Studio UI for prototyping, and how built-in extensions (OpenAIAssistantAgent, McpWorkbench, etc.) simplify tasks【37†L64-L74】.

Lab 23: *Build a Simple 2-Agent Workflow with AutoGen*

- **Task:** Use AutoGen to set up two agents:
- **SocAgent:** Investigates an alert.

- **ReportAgent:** Formats the findings into a summary.
  The SocAgent generates the facts and passes them to ReportAgent. (This can be done by calling `AssistantAgent.run` from within another agent's conversation, or using the new Core API if available.)

- **Outline Code:**

```
from autogen_agentchat.agents import AssistantAgent
from autogen_ext.models.openai import OpenAIChatCompletionClient

soc_agent = AssistantAgent("soc_agent",
OpenAIChatCompletionClient(model="gpt-4o"))
report_agent = AssistantAgent("report_agent",
OpenAIChatCompletionClient(model="gpt-4o"))

soc_result = await soc_agent.run(task="Analyze alert: 'Multiple login
failures.'")
report_input = "SOC findings:\n" + soc_result
report = await report_agent.run(task=f"Write a brief report based on:
\n{report_input}")
print(report)
```

- **Expected:** `report` should be a concise narrative report built from SocAgent's output.

Quiz Q23: *Which AutoGen component lets an agent use OpenAI as if it were an assistant?*
A23: `AssistantAgent` (part of AgentChat) – it wraps an OpenAI client to interact in a conversational manner【37†L29-L38】.

Day 24: Voice & Multi-modal (LiveKit)

Objectives: Explore multi-modal agents: voice interfaces via LiveKit, and mixing text/audio.

Lecture Outline:

- **LiveKit Overview:** LiveKit is an open-source framework for voice/video AI agents【39†L180-L189】. It provides SDKs and cloud for voice assistants (speech-to-text, LLM, text-to-speech).
- **Pipeline:** User voice → Speech-to-Text → LLM (with or without context/tools) → Text-to-Speech. It even supports telephony (real phone calls).
- **Coding Agents:** LiveKit can integrate with coding assistants (e.g., Claude Code)【39†L208-L214】.
- **Use-Case:** Imagine a 24/7 voice SOC assistant: you call a number, speak "Check incidents from last hour", and the system responds verbally.
- **Demo:** We won't fully implement audio, but note that LiveKit's Quickstart can get a dev-agent working in minutes【39†L180-L189】.

Quiz Q24: *What is an advantage of a voice-based AI assistant in security ops?*
A24: It allows hands-free, natural interaction (e.g. during an incident, an on-call engineer can query the system by voice), leveraging speech-to-text and TTS with LLM reasoning.

Day 25: Production Architecture

Objectives: Transition from prototypes to production: systems design, security, scalability.

Lecture Outline:

- **Service Architecture:** Multi-tier design:
- **API Gateway / Load Balancer** in front of services (for scaling & auth).
- **AI Service Layer:** This runs LLM APIs/agents (could be microservices).
- **Data Layer:** Databases (vector DB, KG, logs).
- **UI/UX:** The Streamlit or web frontend behind its own endpoints (with auth).
- **Security Controls:** (OWASP/NIST aligned)
- **Authentication/Authorization:** Only authenticated users/systems can access APIs. Use JWT/OAuth for endpoints.
- **Data Encryption:** In-transit (HTTPS) and at-rest (disk encryption).
- **Prompt Guardrails:** E.g. prefix instructions to ignore harmful content, implement rate-limits, use OpenAI's content filters.
- **Logging & Monitoring:** Log all LLM queries and results (for audit and debugging). Use tools like LangChain's LangSmith for tracing. Monitor latency and cost (API usage metrics).
- **Model Lifecycle:** Implement strict model access (e.g. only approved models), version control for prompts, CI/CD for agent code.
- **Compliance:** Ensure data entering LLM doesn't violate DPDP (no unwanted PII)【45†L102-L111】.
- **Cost Controls:** Use cheaper models for lower-critical tasks; use batching【50†L859-L868】; cache frequent queries; set spending limits (OpenAI's Spend Limits)【50†L829-L838】.
- **Scalability:** Containerize agents (Docker) and use Kubernetes or serverless for dynamic scaling. Use managed services (e.g., SageMaker, Vertex AI) to autoscale.

Quiz Q25: *Name two infra measures to secure an LLM-based API.*
A25: 1. TLS (HTTPS) for all traffic.
2. Authentication (e.g. API tokens, OAuth) for endpoints.
(Others: Rate limiting, input sanitization, secret management, etc.)

Week 6: Advanced Topics & Capstone (Days 26–30)

Day 26: Fine-Tuning vs RAG vs Prompting

Objectives: Understand when to fine-tune a model vs using RAG vs relying on prompt engineering.

Lecture Outline:

- **Fine-Tuning:** Re-trains the model on domain-specific data. Pros: model "knows" data intrinsically; responses more consistent. Cons: expensive, needs large data & time, static knowledge (no real-time update).
- **When to Fine-Tune:** Small, stable knowledge domains (e.g. company policy), or specialized writing style. Not typically needed if RAG covers it. Note: OpenAI now favors **plugins/RAG** over custom fine-tunes (see OpenAI guidance【5†L15-L21】).
- **Prompting & In-context:** For most use-cases, writing good prompts (with examples) is enough.
- **Comparison (from Pinecone):** RAG is more cost-effective and maintains accuracy by retrieving real data【28†L131-L139】. A fine-tuned model may hallucinate if data changes, whereas RAG can always fetch updated info.
- **Practice:** We did not cover large-scale fine-tuning here, but note that frameworks like Hugging Face's 🤗 Trainer, or OpenAI's Fine-tuning API (GPT-3.5) exist.

Quiz Q26: *Why might an enterprise prefer RAG over fine-tuning their own model?*
A26: RAG uses existing foundation models and external data, so it avoids the heavy costs and rigidity of re-training. It lets you update knowledge simply by updating the database【28†L131-L139】.

Day 27: Claude Code and Model Customization

Objectives: Introduce Claude Code (for AI-assisted coding) and model "customization" tools (e.g. OpenAI's functions, plugins).

Lecture Outline:

- **Claude Code:** A tool by Anthropic that assists software developers (like GitHub Copilot). It can write, explain, and debug code. We can use it to build parts of our projects (e.g. generate code for data processing).
- **Use-Case:** For example, ask Claude Code: "Write a Streamlit code snippet to display a table of alerts." It outputs Python code, which we can adapt.
- **OpenAI "Function calling":** (If part of instructor scope) Mention that some models support structured function calls (returning JSON specifying function and args) for safer automation.
- **AutoGen Extensions:** We saw `OpenAIAssistantAgent` and `McpWorkbench`【37†L64-L72】. These let agents integrate with GitHub, Slack, etc. The MCP plugin in AutoGen can allow agents to perform code commits, API calls, etc.

- **Limitations:** Even with Claude Code, always review AI-generated code for logic errors. Run tests. (OWASP LLM02: insecure output can be harmful【31†L133-L140】.)

Quiz Q27: *What is one way Claude Code can help in this course?*
A27: It can auto-generate code templates or snippets (e.g. Python/Streamlit code) from prompts, accelerating development.

Day 28: Prompt-LLM Agents (Claude Code / Cursor)

Objectives: Practice using LLM-based coding assistants (like Claude Code or Cursor) to build our system faster.

Lecture Outline:

- **Tools:** Aside from Claude Code, there's Google's `Cursor` (IDE AI assistant) or VS Code Copilot. These can read our code and generate further code.
- **Workflow:** Example: Use Claude Code via API to generate a query or transform JSON, then test it. Or use Cursor in code editor to build LangChain chains by comments.
- **Demo:** (Instructor may show quick demo generating a Streamlit component or parsing logic via Claude).
- **Caveat:** Always vet the code. AI suggestions may use deprecated libraries or not meet exact requirements.

Lab 28: *AI-assisted Coding*

- **Task:** In your IDE, use an AI assistant (or the `openai.ChatCompletion` with Claude Code model if available) to generate a piece of code. For example:
- Prompt: "Generate Python code to call an n8n webhook and parse its JSON response."
- **Expected:** The assistant should produce a valid code snippet using `requests` or similar.

Quiz Q28: *What is a best practice when using AI-generated code?*
A28: Always review, test, and modify it to ensure it's correct, secure, and uses up-to-date libraries.

Day 29: Enterprise GenAI System Design

Objectives: Architect a full enterprise-grade GenAI system (SOC-focused), combining all learned elements.

Lecture Outline:

- **Architecture Recap:** Present a full diagram (see Mermaid below) of the envisioned system (users, gateways, orchestrators, LLMs, storage, tools). Discuss each component's role and tech choices.
- **Security/Governance:** Revisit OWASP GenAI Top 10【31†L128-L137】: prompt injection defenses, output validation, plugin safety, misuse monitoring. NIST AI RMF encourages documentation, risk assessments, and controls【33†L153-L160】.
- **Compliance (India):** Highlight DPDP Act obligations for LLM services【45†L58-L67】: ensure personal data is used lawfully (consent), implement data minimization (no unnecessary PII in prompts)【45†L102-L111】, and have breach plans (log exposures).

- **Monitoring & MLOps:** Use tools like LangChain LangSmith for tracing queries, Prometheus/Dashboard for performance, and define KPIs (accuracy, latency, cost).
- **Business Alignment:** Tie back to SOC goals: improved response time, reduced false positives, compliance audit readiness.

```
flowchart TD
    UserWeb[User (Analyst) - Web UI / API]
    UserWeb --> API_GW[API Gateway / Auth]
    API_GW --> Orchestrator[AI Orchestrator (LangChain/Agent)]
    subgraph AI_Service [Model & Agents]
      Orchestrator --> LLM[LLM Model Service (GPT-4o/AzureAI/Gemini)]
      Orchestrator --> VGDB[Vector DB (Pinecone/Weaviate)]
      Orchestrator --> RAGSearch[Retrieval + Context]
      Orchestrator --> Tools[Tools/MCP (SIEM, WHOIS, MalwareDB)]
      LLM --> Orchestrator
      VGDB --> Orchestrator
      Tools --> Orchestrator
    end
    API_GW --> DataStore[Data Stores (Logs, KB, KG)]
    DataStore --> Orchestrator
    subgraph UI [Result & Monitoring]
      Orchestrator --> Frontend[Streamlit Dashboard / Chat]
      Orchestrator --> Ticketing[Jira/Ticket System]
      Orchestrator --> Logs[Logging & LangSmith]
      Logs --> Monitoring[Monitoring/Alerts]
    end
```

(Figure: Enterprise GenAI SOC Architecture – Users interact via UI/API, which hits an orchestrator. The orchestrator calls the LLM and retrieval modules. External Tools (SIEM, WHOIS, threat intel DBs) and Vector DBs feed context. Results go back to the user and to monitoring/logging systems.)

Quiz Q29: *What are two key considerations when deploying LLMs in a regulated environment?*
A29: Data privacy (e.g. encrypting sensitive data, not sending PII to external APIs) and auditability (logging inputs/outputs for review)【45†L102-L111】【33†L153-L160】.

Day 30: Capstone Project – AI SOC Analyst

Objective: Deliver an end-to-end GenAI SOC Assistant that covers alert intake, analysis, and action.

Capstone Description:
Build an **"AI SOC Analyst"** system combining all modules:

1. **Input:** A user submits a security alert (via web UI or webhook).
2. **Agent Orchestration:** The orchestrator (LangChain/AutoGen) retrieves relevant policy/docs (RAG) and uses agents:

3. **SOC Agent:** Analyzes alert details, assigns severity, suggests MITRE ATT&CK IDs.

4. **Intel Agent:** Gathers external info (WHOIS, VirusTotal).
5. **Report Agent:** Writes a summary of findings.
6. **Knowledge Graph:** Store known entities/relations (e.g. IP ↔ threat actor). Agent can query KG for context.
7. **Output:** Show results in a Streamlit dashboard (severity badge, MITRE links, action items). Use n8n to also create a Jira ticket if severity ≥ High.
8. **Security:** Use guardrails – e.g., sanitize inputs, log all steps (LangSmith), enforce auth on UI.

Deliverables:

- Source code (Python notebooks/modules) with README.
- Running demo (Streamlit or similar) showing an example alert through the pipeline.
- A short presentation (slides or document) explaining architecture and lessons.

Example Scenario: "User: Investigate IP 203.0.113.5." The system retrieves logs, sees brute-force attempts. WHOIS says APNIC. VirusTotal negative. It answers: "IP 203.0.113.5 is from APNIC (Asia). It had 50 failed SSH login attempts (likely credential stuffing). Recommended actions: block IP, ensure MFA. MITRE IDs: T1110, T1078." The UI displays this, and a Jira ticket is created.

Mini-Projects (Optional):

1. **SOC Alert Summarizer:** A Streamlit app (with LangChain) that takes a SIEM alert log and outputs a structured summary (JSON format).
2. **RAG-based Chatbot:** An enterprise chatbot that answers questions about security policies (uploads company docs into Weaviate/Pinecone and answers queries with citations).
3. **Compliance Assistant:** An LLM agent that checks if a data handling scenario complies with DPDP rules (reads user inputs, cross-checks consent conditions).

Each mini-project should have a short write-up of objectives and output.

Tools and Options (Comparison Tables)

LLM / Model Options:

Model / Service	Provider	Deployment (cloud/self)	Strengths	Weaknesses	Enterprise Fit
GPT-4o	OpenAI	API (AWS, Azure, Google)	High accuracy, broad knowledge	Black-box, cost per token	General-purpose, broad support
Claude-3 Sonnet	Anthropic	API (Claude Cloud)	Good at large texts, safe completions	Slightly higher latency	Document-heavy tasks

Model / Service	Provider	Deployment (cloud/self)	Strengths	Weaknesses	Enterprise Fit
Gemini-1.5	Google	API (Vertex AI)	Multimodal, large context	Newer ecosystem	When multimodal needed
LLaMA-3	Meta	Self-host (inference)	Flexible, fully controlled	Smaller context, needs infra	On-prem/ customizable
Mistral-Instruct	Mistral AI	API / self-host	Strong open-source performance	Smaller ecosystem	Cost-sensitive, privacy needs

Note: Context lengths and pricing are rapidly evolving. Always check latest model specs. As Adobe notes, context handling, latency, and governance are as critical as raw accuracy【19†L117-L126】【19†L144-L151】.

Vector DB Options: Refer to table in Day 12.

Agent Frameworks:

Framework/ Tool	Type	Key Features	Best for
LangChain	Python library	LLMChains, RetrievalQA, Agents (React), deep agents【9†L1548-L1557】	Custom pipelines, RAG apps
LangGraph	JavaScript/ Python	Node-based flow for agents (visual)	Interactive agent orchestrations
AutoGen	Python (MSR)	Multi-agent Core, AgentChat, MCP support【37†L24-L33】	Multi-agent workflows, coding
n8n	No-code Workflows	Visual automation, many integrations 【11†L21-L29】	Glue between APIs (non-dev)
Other	–	Google's Airchat, Meta Visual Chatbot… (we focus on above)	

Cloud Deployment:

Cloud/ Service	GenAI Services	Pros	Cons
AWS	SageMaker, Bedrock	Mature, enterprise-friendly, many models	Cost can be high; vendor lock-in
Azure	Azure OpenAI, Azure ML	Integrated with MS stack, AzureAD auth	Region availability, slightly higher latency

Cloud/ Service	GenAI Services	Pros	Cons
GCP	Vertex AI (Gemini)	Good multi-modal support, strong infra	Fewer LLM options than AWS/Azure
Hybrid/On-Prem	Self-host (Ollama, HuggingFace)	Full data control, possibly cheaper	Requires infra and ops effort

Cost control: Use provider's cost alerts/quotas. E.g., OpenAI allows **spend limits** and Azure/AWS have budgeting tools. Always choose the smallest model that meets accuracy needs【50†L848-L856】.

Security & Governance Checklists

- **OWASP GenAI Top 10:** Ensure defenses against: Prompt Injection (LLM01) by sanitizing inputs; Insecure Output Handling (LLM02) by validating LLM outputs; Training Data Poisoning (LLM03) via vetted data; Model DoS (LLM04) via rate limits; Supply-chain (LLM05) by checking third-party libs; Sensitive Data Exposure (LLM06) by redacting PII; Unsafe Plugins (LLM07) by access controls; Excessive Agency (LLM08) by limiting agent permissions; Overreliance (LLM09) by human review; Model Theft (LLM10) by securing model access【31†L128-L137】.
- **NIST AI RMF:** Document your system's purpose, data flows, and risk mitigations. Conduct regular risk assessments focusing on fairness, safety, privacy【33†L153-L160】. Use NIST's GenAI Profile (AI RMF 2.0) for guidance on generative AI controls.
- **India DPDP Act:** Before any AI process, ensure **Consent** is obtained for data used (especially personal identifiers)【45†L58-L67】【45†L102-L111】. Minimize personal data in prompts/ knowledge bases. Implement rights (e.g. ability to delete user data). Report any data leakage as required.
- **Other:** Comply with CERT-In advisories (e.g., "Best practices against GenAI vulnerabilities")【47†L268-L275】. Regularly review logs for anomalies.

Assessment Quizzes (Example)

(Each week's end could have a short quiz. Example questions have been included above with each day. Instructors can combine them into a 10-question quiz, or use them periodically for checks.)

Recommended Resources

- **LLM Basics & Prompting:** OpenAI docs (Prompting guide), Anthropic docs, Google Gemini guides.
- **RAG & LangChain:** Pinecone's RAG guide【28†L108-L117】; LangChain RAG tutorials (see [9]); LangChain Retriever and DeepAgents docs【9†L1425-L1433】.
- **Vector DB:** Weaviate Docs (Overview【26†L111-L119】), Pinecone docs, Firecrawl vector DB guide【21†L333-L341】.
- **Agents & AutoGen:** AutoGen docs【37†L29-L38】, LangChain Agents docs, LiveKit Docs【39†L180-L189】.
- **Security:** OWASP GenAI Top 10【31†L128-L137】, NIST AI RMF overview【33†L153-L160】, CERT-In AI advisories【47†L268-L275】, DPDP Act guides【45†L58-L67】【45†L102-L111】.

- **Cloud/DevOps:** AWS/Azure/GCP GenAI service docs; OpenAI cost optimization guide【50†L848-L856】.

Each day's lecture notes above reference key sources (citations). Instructors should encourage reading the cited docs for depth. All code snippets can be provided to students as templates in a Git repo.

This comprehensive plan, with objectives, labs, quizzes, code examples, diagrams, and references, gives a clear **roadmap to become a GenAI Architect** specializing in secure, enterprise AI for SOC use-cases.